Propagating systems of dense linear integer constraints

Thibaut Feydy and Peter J. Stuckey

March 31, 2008

Abstract

In interval propagation approaches to solving nonlinear constraints over reals it is common to build stronger propagators from systems of linear equations. This, as far as we are aware, is not pursued for integer finite domain propagation. In this paper we show how we use interval Gauss-Jordan elimination to build stronger propagators for an integer propagation solver. In a similar fashion we present an interval Fourier elimination preconditioning technique to generate redundant linear constraints from a system of linear inequalities. We show how to convert the resulting interval propagators into integer propagators. This allows us to use existing integer solvers. We give experiments that show how these preconditioning techniques can improve propagation performance on dense systems.

1 Introduction

Linear equations and inequalities are one of the most important constraints in any integer finite domain propagation system. Efficient bounds propagation of individual linear constraints is well understood [8] and available in all constraint programming systems. But in other solving approaches systems of linear constraints are not treated individually but together as a system.

Example 1 *Consider the linear equations $x_1 + x_3 - x_4 = 3 \wedge x_1 + x_2 + 2x_3 - x_4 = 4$ with initial domains $[0, 4]$ then individually no propagation is possible, but the equivalent system $x_1 + x_3 - x_4 = 3 \wedge x_2 + x_3 = 1$ can reduce the domains of all variables. Clearly if we can treat the system together there is more scope for propagation*

In linear programming the real relaxation of all integer linear equations and inequalities is treated together by the linear programming algorithm. Even though linear programming based propagators [13] are available in constraint programming toolkits (for example OPL [12] and ECLiPSe [3]), the linear programming propagator does not provide new bounds information for the integer variables involved, rather it is used to bound the objective function (and possibly to do reduced cost variable fixing).

Alternatively one can also use linear programming to build propagators that take into account all linear constraints simultaneously by minimizing and maximizing every variable in turn, and rounding the resulting bounds to integers [14]. Unfortunately this requires repeatedly performing $2n$ LP optimizations, where n is the number of variables, to propagate the linear constraints. This is very expensive compared to the cost of treating the constraints as integer linear propagators.

In this paper we propose preconditioning techniques, based on a whole system of linear constraints, to generate strong linear propagators.

In nonlinear interval solvers, systems of linear equations are treated together as propagators using Gauss-Seidel iteration. A preconditioning step is typically applied first and involves computing a inverse matrix, using Gauss-Jordan elimination or an equivalent procedure [7]. While it is possible to define Gauss-Jordan elimination over integer linear equations, in practice the integer coefficients typically quickly explode making it impractical.

In this paper we explore using a real interval hybrid approach to propagate systems of integer linear equations. Preconditioning of the linear equations is performed using interval arithmetic, and the resulting interval arithmetic propagator is applied to variables (taking into account their integrality). This requires a hybrid interval and integer finite domain solver.

Example 2 *Consider the following constraints which are almost collinear to those of Example 1:* $100x_1+99x_3-101x_4 = 301 \wedge 101x_1+99x_2+199x_3-100x_4 = 399$*, and initial domains [0,4]. Again no propagation is possible. The integer Gauss-Jordan elimination yields* $100 \times 99x_2 + (100 \times 199 - 101 \times 99)x_3 - (100 \times 100 - 101 \times 101)x_4 = 100 \times 399 - 101 \times 301$*, illustrating the increase in coefficients. The interval version of the elimination yields* $x_2 + \widehat{\frac{\mathbf{9901}}{\mathbf{9900}}}x_3 - \widehat{\frac{\mathbf{201}}{\mathbf{9900}}}x_4 = \widehat{\frac{\mathbf{9499}}{\mathbf{9900}}}$ *where* $\tilde{a}$ *represents a "tight" floating point interval around* a*. Both the integer and interval propagators resulting from Gauss-Jordan elimination detect that there is no solution.*

In a similar fashion we present a Fourier elimination scheme using interval arithmetic, which, given an original system of inequalities, allows us to generate safe redundant inequalities. We use it to generate interval arithmetic propagators. Given the exponential behavior of the original Fourier algorithm, we use a partial algorithm of polynomial complexity, generating a linear number of inequalities that propagates well.

We show how we can remove the requirement for using a hybrid solver by weakening the interval linear equations and inequalities produced by the above preconditioning techniques to integer linear inequalities, as illustrated in example 3 below.

Example 3 *The floating point interval equation resulting in Example 2 can be weakened to the integer inequalities* $9899x_2+9900x_3-202x_4 \leq 9500$ *and* $9901x_2 + 9902x_3 - 200x_4 \geq 9498$*. This is at least strong enough to reduce the domains of* x_2 *and* x_3 *to [0,1], and with the original equation* x_1 *obtains [1,4] and* x_4 *obtains [0,2].*

Note that x_4 *is almost irrelevant in these equations since its coefficient is much smaller than the other coefficients. We can remove it altogether,* *using its bounds, to simplify the inequalities without losing much propagation. We obtain* $9899x_2 + 9900x_3 \leq 10308$ *and* $9901x_2 + 9902x_3 \geq 9498$. *We lose no initial propagation with this simplification.*

This paper examines the uses of interval Gauss-Jordan elimination and interval Fourier elimination preconditioning techniques for integer linear equations and inequalities. We show how to achieve this using a hybrid interval and integer propagation solver or even how to map back to a full integer problem. Our experiments show that these techniques can lead to substantial improvements in propagation efficiency. A preliminary version of this paper examining only Gauss-Jordan elimination appeared as [4]

The remainder of the paper is organized as follows. In the Section 2 we introduce notation, and the necessary background for understanding linear constraint propagation, and interval reasoning methods. Section 3 presents an interval scheme for integer systems of equality while the Section 4 presents a similar scheme for system of inequalities. In Section 5 we give experimental results, and conclude in Section 6.

2 Background

2.1 Interval arithmetic

Let $\mathbb{R}$ be the set of real numbers with $\{-\infty, +\infty\}$, and let $\mathbb{F}$ be the subset of $\mathbb{R}$ of the representable floating-point numbers in a given format. Given a real number r, we will use $\downarrow(r)$, $\uparrow(r)$ and $nearest(r)$ to represent the downward rounding, upward rounding and rounding to the nearest, on $\mathbb{F}$.

An interval has the form $\boldsymbol{x} = [\underline{x}, \overline{x}]$ where $\underline{x}$ and $\overline{x}$ are two real numbers and $\underline{x} \leq \overline{x}$. We will use $\mathbb{IR}$ to represent the set of intervals over reals, and $\mathbb{IF}$ to represent the set of intervals with endpoints from $\mathbb{F}$. Both are closed under intersection.

Given an interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$ then $\inf \boldsymbol{x} = \underline{x}$ and $\sup \boldsymbol{x} = \overline{x}$ define the lower and upper bound of $\boldsymbol{x}$ respectively, while the absolute value is defined as $|\boldsymbol{x}| = \max\{|a|, |b|\}$.

Given an interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$, let $floor(\boldsymbol{x}) = \lfloor \underline{x} \rfloor$ be the largest integer bounding $\boldsymbol{x}$ from below and $ceil(\boldsymbol{x}) = \lceil \overline{x} \rceil$ be the smallest integer bounding $\boldsymbol{x}$ from above.

Given an interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$ let $\operatorname{mid} \boldsymbol{x}$ be the approximated midpoint of $\boldsymbol{x}$:

$$\operatorname{mid} \boldsymbol{x} = nearest(\frac{\overline{x} + \underline{x}}{2})$$

Note that for unbounded intervals, any value in the interval can be used as an approximate midpoint. The functions *ceil* and *floor* are not defined for unbounded intervals, however we will only use them on bounded intervals in the rest of the paper.

Given two intervals $\boldsymbol{x}$ and $\boldsymbol{y}$ the float interval operator $\diamond$ associated to the real operator $\diamond$ is defined by:

$$\boldsymbol{x} \diamond \boldsymbol{y} = \bigcap_{\boldsymbol{z} \in \mathbb{IF}} \{\boldsymbol{z} \mid \forall x \in \boldsymbol{x}, \forall y \in \boldsymbol{x}, x \diamond y \in \boldsymbol{z}\}$$

As an example, the multiplication operator is defined by:

$$[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] = [\downarrow \min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \uparrow \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]$$

We will use $\cap$ and $\cup$ to represent respectively the union and intersection of two intervals.

A *box* $\boldsymbol{d}$ is a pair $[\underline{d}, \overline{d}]$ consisting of two real column vectors $\underline{d}$ and $\overline{d}$ of length n with $\underline{d} \leq \overline{d}$. A box is indentified with the (nonempty) set of points it contains $\boldsymbol{d} = \{d \in \mathbb{R}^n \mid \underline{d} \leq d \leq \overline{d}\}$. A *thin box* $[d, d]$ contains a unique point d and we can extend operators to apply to mixes of intervals and scalars by identifying the scalar d with the thin box $[d, d]$ and using the interval operator.

Example 4 *Let $\boldsymbol{x}$ be the interval $[-1, 1]$, let y be the scalar 10, and let $\boldsymbol{z}$ be $\boldsymbol{x}/y$. Then:*

$$\begin{array}{rcl}
\boldsymbol{z} & = & \boldsymbol{x}/y = [-1,1]\,/\,[10,10] = [\downarrow -0.1, \uparrow 0.1] \\
\inf \boldsymbol{z} & = & \underline{\boldsymbol{z}} = \downarrow -0.1 \\
floor(\boldsymbol{z}) & = & \lfloor\ \downarrow -0.1 \rfloor = -1 \\
\text{mid}\,(\boldsymbol{z}) & = & nearest(\dfrac{\downarrow -0.1 + \uparrow 0.1}{2}) = 0
\end{array}$$

Interval arithmetic [10] was designed to tackle two problems of numerical analysis: uncertainty of data and roundoff error. It has the advantage of being a sound approximation of problems over $\mathbb{R}$, and interval analysis methods, based on iterative contraction of intervals, are easily implemented in a constraint programming framework.

The following standard result gives the basis of interval arithmetic.

Theorem 1 (Fundamental Theorem of Interval Arithmetic [10]) *Given an arithmetic expression* f*, and an interval vector* $\boldsymbol{x}$*, for all real vector* $v \in \boldsymbol{x}$*, the real evaluation of* f *at* v *is contained in the interval evaluation* $f(\boldsymbol{x})$*.*

2.2 Gauss-Jordan elimination

Gauss-Jordan elimination is a version of Gaussian elimination which solves a system of equations $AX = B$ by applying elementary row operations in order to transform the coefficients matrix to a reduced row echelon form. Elementary row operations are linear transformations on a matrix that do not change its solution set (neglecting rounding errors). If applied on a square matrix A, it can be used to compute its inverse A^{-1}: a sequence of transformation $T_1, \ldots, T_n$ is applied to the system $AX = I$ to transform the coefficient matrix into the identity matrix: $IX = T_1 \ldots T_n I$

Each step of the elimination process (or equivalently each transformation matrix T_i), transforms a column c_i into a zeroed column, except the diagonal element which is equal to 1. This *pivot* operation can be decomposed into three operations:

- swapping two rows to ensure than the diagonal element a_{ii} is not zero.
- multiplying the row c_i by $1/a_{ii}$ so that $a_{ii} = 1$.

- for each row $j \neq i$, subtracting a_{ji} times the row i so that a_{ij} is now zeroed.

Swapping rows to ensure that the eliminated coefficient a_{ii} is the largest possible provides good numerical stability compared to a more naive swapping. Columns swapping can also be used with/in place of row swapping to choose the next pivot. However, while row swapping corresponds to interchanging equations, column swapping corresponds to interchanging variables. Thus the computed matrix will be the inverse of A scrambled by column. The original order can be restored with minimal bookkeeping.

2.3 Gauss-Seidel

Gauss-Seidel is an iterative method to solve linear system of equations. Given a system $Ax = b$, where A is an $n \times n$ matrix, the Gauss-Seidel algorithm starts from an initial guess of values for x, $x^{(0)}$. In iteration k the algorithm It then repeatedly computes a new value $x_1^{(k)}, \ldots, x_n^{(k)}$ for each variable in x in term using the most up to date values available using the following formula:

$$x_i^{(k)} := \frac{b - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)}}{a_{ii}}$$

The algorithm continues until convergence or some iteration limit is reached. In order to improve or accelerate the convergence of the algorithm, a transformation is usually applied to the system before beginning the iterations. This is generally done by multiplying the original system $Ax = b$ by a suitable matrix, or *preconditioner*, P. to obtain the system $PAx = Pb$. A preconditioner aims to make the matrix diagonally dominant, that is where $\forall i \forall j. |(PA)_{ii}| > (PA)_{ij}$ and $\forall i \forall j. |(PA)_{ii}| > (PA)_{ji}$.

The Interval Gauss-Seidel method solves interval system of equations $\boldsymbol{Ax} = \boldsymbol{b}$ using the same algorithm as Gauss-Siedel but with interval arithmetic. Hence starting from initial (large) intervals $\boldsymbol{x}^{(0)}$ the iterative process computes

$$\boldsymbol{x}_i^{(k)} := \boldsymbol{x}_i^{(k-1)} \cap \frac{\boldsymbol{b} - \sum_{j<i} \boldsymbol{a}_{ij} \boldsymbol{x}_j^{(k)} - \sum_{j>i} \boldsymbol{a}_{ij} \boldsymbol{x}_j^{(k-1)}}{\boldsymbol{a}_{ii}} \tag{1}$$

Note that this equation does nothing when $0 \in \boldsymbol{a}_{ii}$ since the left hand side of the intersection is the widest possible interval.

In infinite precision, Interval Gauss-Seidel would converge to the hull of solutions on a linear system with a diagonally dominant matrix [11], while in practice a slightly larger hull is obtained due to outward rounding. As in the non-interval case a preconditioner is used to improve convergence on general systems. The most widely used preconditioner, often considered optimal, is the midpoint inverse of A [7]:

$$\tilde{\boldsymbol{A}}^{-1} = (mid(\boldsymbol{a}_{ij}))^{-1}$$

$\tilde{\boldsymbol{A}}^{-1}$ can be computed approximatively using floating-point inversion algorithms, such as Gauss-Jordan elimination or LU decomposition. Preconditioning involves $O(n^3)$ interval multiplications, thus the resulting system will have a wider hull than the original one. An inconsistent system of equation may become consistent after preconditioning due to the widening, however *completeness* is guaranteed: all solutions of the original system are solutions of the preconditioned one (comes from Definition 1).

The coefficient matrix $\tilde{\boldsymbol{A}}^{-1}\boldsymbol{A}$ resulting from this preconditioning is a matrix whose diagonal elements are centered around one, while non-diagonal elements are centered around zero. When the initial coefficient matrix $\boldsymbol{A}$ is a floating-point matrix, the width of the coefficients after preconditioning results from rounding during computations. We may then expect these intervals to be tight and we will refer to them as *quasi-zeros* for non-diagonal coefficients and *quasi-ones* for diagonal coefficients. We will refer to the resulting matrix as a quasi-identity matrix.

2.4 Linear constraint propagators

Constraint propagation engines maintain a domain of possible values for each variable, and apply propagators to narrow the domain of possible values. Propagators are functions from domains of possible values to domains of possible values. Propagators for linear constraints in finite domain solvers typically perform bounds($\mathbb{R}$) propagation [2]. This relies only on the bounds of the variables, and furthermore does not make use of integrality, since doing so, bounds($\mathbb{Z}$) propagation, is NP-hard. Since propagation only relies on bounds we can represent a domain of possible value for each variable $x_1, \ldots, x_n$ as a box $\boldsymbol{d} = \boldsymbol{d}_1 \times \cdots \times \boldsymbol{d}_n$ where $x_i \in \boldsymbol{d}_i$. In this framework a propagator for a constraint c is a function $c :: \mathbb{IR}^n \to \mathbb{IR}^n$.

The bounds($\mathbb{R}$) propagator for $c \equiv a_1x_1 + \cdots a_nx_n \leq a_0$ is defined as $c(\boldsymbol{d}) = \boldsymbol{d}'$ where

$$\begin{array}{lcll}
\boldsymbol{d}'_i & := & \left[\inf \boldsymbol{d}_i, \min\{\sup \boldsymbol{d}_i, \sup(\frac{a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j}{a_i})\}\right] & \text{if } a_i > 0 \\
\boldsymbol{d}'_i & := & \left[\max\{\inf \boldsymbol{d}_i, \inf(\frac{a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j}{a_i})\}, \sup \boldsymbol{d}_i\right] & \text{if } a_i < 0 \\
\boldsymbol{d}'_i & := & \boldsymbol{d}_i & \text{if } a_i = 0
\end{array}$$

Bounds($\mathbb{R}$) propagation for the equality $c \equiv a_1x_1 + \cdots a_nx_n = a_0$ just combines the propagators for $c_{\leq} \equiv a_1x_1 + \cdots a_nx_n \leq a_0$ and $c_{\geq} \equiv -a_1x_1 - \cdots a_nx_n \leq -a_0$. The propagator for c is defined as $c(\boldsymbol{d}) = c_{\leq}(\boldsymbol{d}) \cap c_{\geq}(\boldsymbol{d})$, or equivalently (for $a_i \neq 0$)

$$\boldsymbol{d}'_i := \boldsymbol{d}_i \cap \frac{a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j}{a_i} \tag{2}$$

We can extend bounds($\mathbb{R}$) to interval constraints $\boldsymbol{a}_1x_1 + \cdots \boldsymbol{a}_nx_n \leq \boldsymbol{a}_0$ and $\boldsymbol{a}_1x_1 + \cdots \boldsymbol{a}_nx_n = \boldsymbol{a}_0$ in the natural way. Note that the interval bounds($\mathbb{R}$) propagator is completely analogous to that shown in Equation 1, where $\boldsymbol{d}$ takes the role of $\boldsymbol{x}$. Hence bounds($\mathbb{R}$) propagation of equalities can be considered as a form of interval Gauss-Seidel computation. The differences arise in the freedom that bounds propagation has in picking which propagator to run next, wherease Gauss-Seidel executes them in a particular order.

In this paper we will be interested in the case of bound($\mathbb{R}$) propagation over integer linear constraints. This only requires a small addition. After each bounds propagation, the endpoints of each domain of an integer variable x_i are rounded inward to the nearest integer: hence $\boldsymbol{d}_i := [\lceil \inf \boldsymbol{d}_i \rceil, \lfloor \sup \boldsymbol{d}_i \rfloor]$ for each integer x_i.

A domain $\boldsymbol{d}$ is a *fixpoint* for constraint c if $c(\boldsymbol{d}) = \boldsymbol{d}$. A domain is a fixpoint for a set of constraints $\mathcal{C}$ if it is a fixpoint for all of them.

Example 5 *Consider the integer variables x_1, x_2, x_3, with the domains $\boldsymbol{d}_1 = \boldsymbol{d}_2 = \boldsymbol{d}_3 = [-10, 10]$, and the following linear constraints:*

$$C_1 : \begin{bmatrix} 5 & 3 & 4 \\ -1 & 2 & -2 \\ 1 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 7 \\ -2 \end{bmatrix}$$

Applying the propagator for $c_{\leq} \equiv 5x_1 + 3x_2 + 4x_3 \leq 0$ we determine $\sup((a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j)/a_i)$ is $(0 - 3 \times -10 - 4 \times -10)/5 = 70/5 = 14$ for $i = 1$, $(0 - 5 \times -10 - 4 \times -10)/3 = 90/3 = 30$ for $i = 2$, and $(0 - 5 \times -10 - 3 \times -10)/4 = 80/4 = 20$ for $i = 3$. Since these bounds are smaller than the current value 10 there is no change. Applying the propagator for $c_{\geq} \equiv -5x_1 - 3x_2 - 4x_3 \leq 0$ we determine $\inf((a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j)/a_i)$ is $(0 - (-3) \times 10 - (-4) \times 10)/-5 = 70/-5 = -14$ for $i = 1$, $(0 - (-5) \times 10 - (-4) \times 10)/-3 = 90/-3 = -30$ for $i = 2$, and $(0 - (-5) \times 10 - (-3) \times 10)/-4 = 80/-4 = -20$ for $i = 3$. Again none of these bounds are greater than the current bound -10 so there is no change. Propagation for the other equations likewise does not change the domain $\boldsymbol{d}$. Hence $\boldsymbol{d}$ is a fixpoint of C_1.

Applying the midpoint inverse preconditioner yields the interval constraint system:

$$C_1' : \begin{bmatrix} \widehat{\mathbf{1}} & \widehat{\mathbf{0}} & \widehat{\mathbf{0}} \\ \widehat{\mathbf{0}} & \widehat{\mathbf{1}} & \widehat{\mathbf{0}} \\ \widehat{\mathbf{0}} & \widehat{\mathbf{0}} & \widehat{\mathbf{1}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -\widehat{\mathbf{7}} \\ \widehat{\mathbf{5}} \\ \widehat{\mathbf{5}} \end{bmatrix}$$

In double precision floating-point arithmetic, each interval $\widehat{\mathbf{a}}$ of this system has a width smaller than 10^{-10}. Given any reasonable domains, these constraints fix the values of x_1, x_2 and x_3 during the first iteration to -7, 5 and 5 respectively.

3 Gauss-Jordan elimination

We can use Gauss-Jordan elimination to precondition systems of linear integer equalities to improve the propagation strength. Note that the use of interval arithmetic is required here because using integer/rational arithmetic causes the coefficients to explode.

3.1 Rectangular systems

Many constraint programming problems contain linear equalities, however the number of linear constraints may be less than the number of variables involved in these constraints. In this case only a partial Gauss-Jordan elimination [1] is performed, which may still lead to better propagation. That partial elimination consists of eliminating as many variables as possible by applying a square pseudo inverse to the system (see Example 6).

Given an $m \times n$ matrix A, with m the number of equations and $n \geq m$ the number of variables, the result of the preconditioning of a system $Ax = b$ is a system $(\boldsymbol{I}_m \, \boldsymbol{A}')\, x = \boldsymbol{b}'$, where $\boldsymbol{I}_m$ is a quasi-identity matrix. This is a set of interval equations which can be posted to the constraint system as new propagators. Note these constraints never eliminate a solution of the original system $Ax = b$, but may admit more solutions than the original system due to inaccuracies in floating point arithmetic. Hence we add these new interval propagators in addition to the original propagators.

Example 6 *Consider the preconditioning of the last two equations of the constraint C_1:*

$$C_2 : \begin{bmatrix} -1 & 2 & -2 \\ 1 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ -2 \end{bmatrix}$$

Let us compute the pseudo inverse of the coefficient matrix:

$$\begin{bmatrix} -1 & 2 & -2 \\ 1 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

we pick $a_{1,3}$ as the first pivot, so we do a column permutation:

$$\begin{bmatrix} -2 & 2 & -1 \\ 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_{31} & x_{32} \\ x_{21} & x_{22} \\ x_{11} & x_{12} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

we scale the first row by $1/a_{11} = -1/2$:

$$\begin{bmatrix} 1 & -1 & 0.5 \\ 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_{31} & x_{32} \\ x_{21} & x_{22} \\ x_{11} & x_{12} \end{bmatrix} = \begin{bmatrix} -0.5 & 0 \\ 0 & 1 \end{bmatrix}$$

we subtract $a_{12} = 2$ times the first row from the second:

$$\begin{bmatrix} 1 & -1 & 0.5 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{31} & x_{32} \\ x_{21} & x_{22} \\ x_{11} & x_{12} \end{bmatrix} = \begin{bmatrix} -0.5 & 0 \\ 1 & 1 \end{bmatrix}$$

we now pivot on $a_{22} = 1$. Since it is already on the diagonal and equal to 1, we just subtract $a_{12} = -1$ times the second row from the first.

$$\begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{31} & x_{32} \\ x_{21} & x_{22} \\ x_{11} & x_{12} \end{bmatrix} = \begin{bmatrix} 0.5 & 1 \\ 1 & 1 \end{bmatrix}$$

Note that we can perform all the above computation using floating point arithmetic rather than safe interval arithmetic, since we are seeking only a preconditioner, rather than an accurate inverse.

We then apply the preconditioner to the system using interval arithmetic *to get a safe preconditioned system C_2':*

$$\begin{bmatrix} 0.5 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 2 & -2 \\ 1 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 7 \\ -2 \end{bmatrix}$$

$$\begin{bmatrix} \widehat{\mathbf{1}} & \widehat{\mathbf{0}} & \widehat{\mathbf{0.5}} \\ \widehat{\mathbf{0}} & \widehat{\mathbf{1}} & \widehat{\mathbf{0}} \end{bmatrix} \begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} \widehat{\mathbf{1.5}} \\ \widehat{\mathbf{5}} \end{bmatrix}$$

The original constraints will again not perform any pruning on the domain $\boldsymbol{d}$ *of Example 5 whereas the redundant constraints introduced by C_2' will create a fixpoint $\boldsymbol{d}'$ where $\boldsymbol{d}_1' = [-9, 9]$, the $\boldsymbol{d}_3 = [-3, 6]$, and $\boldsymbol{d}_2 = [5, 5]$. Any further changes in the domain of x_1 and x_3 will strong propagate to the other variable using the first interval equality.*

Note however that partial Gauss-Jordan elimination can result in costly propagators providing weak information if applied to a system of m equations and n variables where $n \gg m$. In the worst case, performing a partial Gauss-Jordan elimination on such a system results in a system of m equations, each with $(n - m) + 1$ non quasi-zero coefficients. If $n \gg m$ and if the original equations each involve on average k variables where $k \ll n$, it is unlikely that we will gain stronger information from preconditioning, while each resulting propagator will have an execution cost in $O(n - m)$ as opposed to $O(k)$ on average for the original propagators.

Example 7 *Consider the system $x_1 + x_2 + x_3 = 0 \wedge x_1 + x_4 + x_5 = 0 \wedge x_5 + x_6 + x_7 = 0$. Preconditioning this system results in the system $x_1 + x_2 + x_3 = 0 \wedge x_4 + x_5 - x_2 - x_3 = 0 \wedge x_6 + x_7 - x_4 + x_2 + x_3 = 0$. No information is gained from the resulting, more expensive, propagators.*

Finally we should discuss the possibility of overconstrained systems. If $n < m$ then preconditioning the system $Ax = b$ results in a system $\begin{pmatrix} \boldsymbol{I}_n \\ \boldsymbol{Z} \end{pmatrix} x = \boldsymbol{b}'$, where $\boldsymbol{I}_n$ is a quasi-identity matrix and $\boldsymbol{Z}$ is matrix of quasi-zeros. This may either immediately result in unsatisfiability being detected or if the original system was linearly dependent results in interval equations of the form $\widehat{\mathbf{0}}x_1 + \cdots + \widehat{\mathbf{0}}x_n = \widehat{\mathbf{0}}$. No useful propagation is ever likely to occur from such constraints so they can be omitted.

3.2 Gauss-Jordan elimination and inequalities

Linear inequalities can be used during Gauss-Jordan elimination, since a linear inequality can be written as an equality constraint by adding a slack variable. However, this is usually less efficient as an inequality constraint can be satisfied (and thrown away) before its variables are ground, which is not true for an equality constraint. Furthermore, using an inequality for pivoting makes the slack variable of this inequality appears in other constraints, which will almost certainly provide very weak propagation.

However, it is still possible to efficiently apply Gauss-Jordan elimination to a system with linear equalities and inequalities as long as inequalities are not chosen for pivoting.

Let us consider a system $Ax = b \wedge Cx \leq d$ of m_A equalities and m_C inequalities. We can rewrite this system by introducing m_C slack variables s:

$$\begin{pmatrix} I_{m_C} & C \\ 0 & A \end{pmatrix} \begin{pmatrix} s \\ x \end{pmatrix} = \begin{pmatrix} d \\ b \end{pmatrix}$$

where I_{m_C} is the $m_C \times m_C$ identity matrix. This system has the form of a general system of equations after m_C pivoting steps, and we can start variable elimination at the $m_C + 1$ row. The result of the next m_A eliminations is:

$$\begin{pmatrix} I_{m_C} & \boldsymbol{Z} & \boldsymbol{C}' \\ 0 & \boldsymbol{I}_{m_A} & \boldsymbol{A}' \end{pmatrix} \begin{pmatrix} s \\ x \end{pmatrix} = \begin{pmatrix} \boldsymbol{d}' \\ \boldsymbol{b}' \end{pmatrix}$$

where $\boldsymbol{I}_{m_A}$ is a quasi-identity matrix and $\boldsymbol{Z}$ is a matrix of quasi-zeros. We can then go back to a reduced system with both equalities and inequalities by removing the unchanged slack variables coefficients submatrix.
$(\boldsymbol{Z}\,\boldsymbol{C}')\,x \leq \boldsymbol{d}' \wedge (\boldsymbol{I}_{m_A}\,\boldsymbol{A}')\,x = \boldsymbol{b}'$.

Example 8 *Consider the system $C_3 = C_2 \wedge 5x_1 + 3x_2 + 4x_3 \leq 0$. We can transform this system into a system of linear equalities by adding a slack variable s:*

$$C_4 : \begin{bmatrix} 1 & 5 & 3 & 4 \\ 0 & -1 & 2 & -2 \\ 0 & 1 & -1 & 2 \end{bmatrix} \begin{bmatrix} s \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 7 \\ -2 \end{bmatrix}$$

The preconditioner computed by following the sames steps as in Example 6 is:

$$\begin{bmatrix} 1 & -5 & -7 \\ 0 & 0.5 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

A partial elimination performed on C_4 gives us the system

$$C_4' : \begin{bmatrix} 1 & \widehat{\mathbf{0}} & \widehat{\mathbf{0}} & \widehat{\mathbf{3}} \\ 0 & \widehat{\mathbf{1}} & \widehat{\mathbf{0}} & \widehat{\mathbf{0.5}} \\ 0 & \widehat{\mathbf{0}} & \widehat{\mathbf{1}} & \widehat{\mathbf{0}} \end{bmatrix} \begin{bmatrix} s \\ x_3 \\ x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} \widehat{-\mathbf{21}} \\ \widehat{\mathbf{3}} \\ \widehat{\mathbf{5}} \end{bmatrix}$$

Note that since we don't use the slack variables for pivoting they will appear in the same order as originally while other variables may have been permuted. Since s is a non-negative slack variable, we replace the first equality by an inequality, leading to the constraint $C_4'' = C_2' \wedge \widehat{\mathbf{3}}x_1 \leq \widehat{-\mathbf{21}}$. Again this gives strong information, with the domain of x_1 reduced to $[-9, -7]$, the domain of x_3 reduced to $[5, 6]$ and x_2 fixed to 5.

3.3 Quasi-zeros elimination

The result of Gauss-Jordan elimination on a system of equations $Ax = b$ with m equations and $n \geq m$ variables is a system $(\boldsymbol{I}_m\,\boldsymbol{A}')\,x = \boldsymbol{b}'$ where $\boldsymbol{I}_m$ is a $m \times m$ quasi-identity matrix. In the case of a floating-point matrix $A \in \mathbb{F}^{n\times m}$, we may expect the quasi-zeros of $\boldsymbol{I}_m$ to be very tight intervals, as they are the result of outward roundings performed during Gauss-Jordan elimination.

Given an interval equation $c \equiv \sum_{i=1\ldots n} \boldsymbol{a}_i x_i = \boldsymbol{a}_0$, let $\mathcal{Q}$ be the subset of $\{1, \ldots, n\}$ such that $\{\boldsymbol{a}_i \,|\, i \in \mathcal{Q}\}$ are quasi-zeros.

Rewriting the propagator for the interval equation (Equation 2) to separate out quasi-zeroes we obtain. The propagator for c is defined as $c(\boldsymbol{d}) = \boldsymbol{d}'$ where (for $i \notin \mathcal{Q}$)

$$\boldsymbol{d}_i' := \boldsymbol{d}_i \cap \frac{\boldsymbol{a}_0 - \sum_{j\notin\mathcal{Q}, j\neq i} \boldsymbol{a}_j \boldsymbol{d}_j - \sum_{j\in\mathcal{Q}, j\neq i} \boldsymbol{a}_j \boldsymbol{d}_j}{\boldsymbol{a}_i} \tag{3}$$

Given the assumption that the quasi-zeros are tight and that we have reasonable initial bounds for each variable, then:

$$\left|\frac{\sum_{k\in\mathcal{Q}} \boldsymbol{a}_k \boldsymbol{d}_k}{\boldsymbol{a}_i}\right| \ll 1.$$

Hence we can replace this expression by its initial value, $\boldsymbol{v} = \frac{\sum_{k\in\mathcal{Q}} \boldsymbol{a}_k \boldsymbol{d}_k^{init}}{\boldsymbol{a}_i}$ where $\boldsymbol{d}^{init}$ is the original domain of variables. The result is the interval equation $\sum_{i=1\ldots n, i\notin\mathcal{Q}} \boldsymbol{a}_i x_i = \boldsymbol{a}_0 - \boldsymbol{v}$. Thus the variables with quasi-zero coefficients are eliminated.

3.4 Going back to integers

Using an external interval solver may be an issue for FD solvers, for efficiency or portability considerations. Obviously, interval constraints need to be available, at least for linear equality constraints. One also needs a way to channel information between the finite domain and the interval solvers. Secondly, whereas it is easy to write portable rounding functions for the arithmetic operators used for the Gauss-Jordan elimination, this is not true for other mathematical functions, making interval solvers more hardware dependent than most FD solvers. And finally, integer operations and constraints are still faster than their floating-point or interval counter-part. That is why we present in this paper a way to use Gauss-Jordan elimination as a preprocessing step leading to posting redundant *integer* constraints.

Given an interval linear equation $c \equiv \sum \boldsymbol{a}_i x_i = \boldsymbol{a}_0$, where the variables x_i are n integers taking values in the box $\boldsymbol{d} = \boldsymbol{d}_1 \times \cdots \times \boldsymbol{d}_n$, and given $2n$ integers $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$, the constraint $a_0 + \sum a_i x_i \leq a_0 \wedge \sum b_i x_i \geq b_0$ where $a_0 = \mathit{ceil}(\boldsymbol{a}_0 + \sum (a_i - \boldsymbol{a}_i)\boldsymbol{d}_i)$ and $b_0 = \mathit{floor}(\boldsymbol{a}_0 - \sum (b_i - \boldsymbol{a}_i)\boldsymbol{d}_i)$ is a safe approximation of c. Indeed $\forall x \in \boldsymbol{d}$, $\sum a_i x_i - a_0 \leq \sum \boldsymbol{a}_i x_i - \boldsymbol{a}_0 \leq \sum b_i x_i - b_0$.

In order to get a tight approximation for the left inequality, the values $\{a_1, \ldots, a_n\}$ are chosen so that the difference $\mathit{diff}_l(x) = \sum \boldsymbol{a}_i x_i - \boldsymbol{a}_0 - (\sum a_i x_i - a_0)$ is small for $x \in \boldsymbol{d}$.

$$\begin{aligned}\mathit{diff}_l(x) &= \sum (\boldsymbol{a}_i - a_i)x_i - \boldsymbol{a}_0 + \mathit{ceil}(\boldsymbol{a}_0 + \sum (a_i - \boldsymbol{a}_i)\boldsymbol{d}_i) \\ &\approx \overline{\boldsymbol{a}}_0 - \boldsymbol{a}_0 + \sum ((\boldsymbol{a}_i - a_i)x_i - \mathit{ceil}(\boldsymbol{a}_i - a_i)\boldsymbol{d}_i))\end{aligned}$$

This last expression is minimized over $\boldsymbol{d}$ by choosing each a_i as the integer value $k \in [\lfloor \boldsymbol{a}_i \rfloor, \lceil \boldsymbol{a}_i \rceil]$ which minimizes

$$\min(|(k - \boldsymbol{a}_i)\boldsymbol{d}_i - ceil((k - \boldsymbol{a}_i)\boldsymbol{d}_i)|).$$

In order to avoid overflow when going back to integers, we may need to multiply the original equation $\sum \boldsymbol{a}_i x_i = \boldsymbol{a}_0$ by a suitable value β:

$$\sum \boldsymbol{a}'_i x_i = \boldsymbol{a}'_0 \text{ where } \boldsymbol{a}'_i = \beta \boldsymbol{a}_i,\ i \in 0, \dots, n$$

However, we choose the largest β which does not cause overflow, if possible $\beta \gg 1$, in order to lose as little information as possible when rounding to integers values, as illustrated by the following example.

A similar reasoning is applied to choose the coefficients b_i for the right inequality $\sum b_i x_i \geq b_0$ similarly.

Example 9 *Consider the preconditioned system C'_2 of example 6, with the initials domains $[-10, 10]$:*

$$C'_2 : \begin{bmatrix} \widehat{\mathbf{1}} & \widehat{\mathbf{0}} & \widehat{\mathbf{0.5}} \\ \widehat{\mathbf{0}} & \widehat{\mathbf{1}} & \widehat{\mathbf{0}} \end{bmatrix} \begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} \widehat{\mathbf{1.5}} \\ \widehat{\mathbf{5}} \end{bmatrix}$$

The equation $\widehat{\mathbf{0.5}}x_1 + \widehat{\mathbf{1}}x_3 = \widehat{\mathbf{1.5}}$ rounded to integers provides the inequalities $x_1 + x_3 \geq -4$ and $x_1 + x_3 \leq 7$, which do not prune the domains of x_1 and x_3. If we first multiply the equality terms by, for example, 100, we get the new equality $\widehat{\mathbf{50}}x_1 + \widehat{\mathbf{100}}x_3 = \widehat{\mathbf{150}}$. This equality provides the inequalities $50x_1 + 100x_3 \geq 149$ and $50x_1 + 100x_3 \leq 151$. This information is strong enough to get the same pruning as the original equation: at fixpoint the domain of x_1 is reduced to $[-9, 9]$ while the domain of x_3 is reduced to $[-3, 6]$. In the same way, for the second equation we get $x_2 \geq 4 \wedge x_2 \leq 6$ with the original equation and $100x_2 \geq 499 \wedge 100x_2 \leq 501$ (i.e. $x_2 = 5$) if we multiply the original equation by 100.

4 Partial Fourier elimination

Linear inequalities are even more common than linear equalities in FD solving. So far, we have seen how to generate strong propagators from a set of equalities using Gauss-Jordan elimination. Elimination schemes, for example Fourier elimination [5] are also available for systems of inequalities, and an interval arithmetic implementation of such a scheme allows us to generate sound propagators, in a fashion similar to what we have done with equalities.

While Gauss-Jordan elimination generates only m equalities given m original constraints, Fourier elimination may lead to the generation of an exponential number of inequalities on general systems. This makes it unusable in practice, especially as we are trying to generate propagators, which are going to run until the problem is solved. As our objective is not to use the Fourier algorithm as a solver but only as a preconditioner that provides valid redundant inequalities, we weaken it by keeping a finite pool of constraint during generation.

4.1 Interval Fourier elimination

Given two inequalities $c_1 \equiv \sum a_i x_i \leq a_0$ and $c_2 \equiv \sum b_i x_i \leq b_0$, with $a_k > 0$ and $b_k < 0$, the *elimination* of x_k from these inequalities produces the inequality $c_{1\times 2,k} \equiv \sum_{i\neq k} (a_i + b_i(-a_k/b_k))x_i \leq a_0 + b_0(-a_k/b_k)$.

Given a set of inequalities $\mathcal{C}$, let $\mathcal{C}_i^+$ be the subset of all inequalities with a strictly positive i-th coefficient and $\mathcal{C}_i^-$ the subset of all inequalities with strictly negative i-th coefficient. $\mathcal{C}_i^0$ is the subset of all inequalities of $\mathcal{C}$ that do not contain x_i. A *Fourier elimination step* consists of performing elimination on all pairs from $\mathcal{C}_i^+ \times \mathcal{C}_i^-$, for a given i, which generates a set of new inequalities $\mathcal{C}_i^+ \otimes \mathcal{C}_i^-$.

The full Fourier elimination algorithm can then be defined as an iteration over the n variables:

- $\mathcal{C}_0 = \mathcal{C}$
- $\forall i \in 1 \ldots n, \mathcal{C}_i = (\mathcal{C}_{i-1})^0 \cup (\mathcal{C}_{i-1})_i^+ \otimes (\mathcal{C}_{i-1})_i^-$

While the Fourier elimination algorithm is in general used to find solutions to a system of inequalities we consider it as an inequality generator, from which we extract good propagators.

The Fourier elimination algorithm performed using floating point arithmetic is in practice neither sound nor complete rounding-errors during elimination can produce systems of inequalities which reject some solutions of the original system or accept values which violate the original systems. Applying the same reasoning to a system of interval inequalities, using interval arithmetic, produces on the other hand a complete system of inequalities. Working with intervals, a variable may not be completely eliminated anymore. Indeed applying a step similar to a Fourier elimination step but with interval arithmetic produces inequalities with one or more coefficients *containing* zero (coefficients are now intervals). Whereas in a Fourier elimination step, inequalities are split between positive, negative and zeros coefficients for a given variable, in a quasi-elimination step inequalities are split between strictly positive, strictly negative and quasi-zeros coefficients for that same variable. For a given constraint, no further elimination is possible on variables with quasi-zeros coefficients.

Interval Fourier elimination leads to safe constraints (that are guaranteed to be logical consequences of the original system of inequalities). Just as with Gauss-Jordan elimination, we can use quasi-zero elimination to eliminate quasi-zeros and then convert these back to integer inequalities in a manner analogous to that described in Section 3.4

Example 10 *Consider the following inequalities where the domains of the variables are* $[-10, 10]$*:*

$$\begin{array}{cccccccc}
x & & & & +u & \leq & 1 \\
-3x & +y & & +2z & & \leq & 3 \\
-3x & & -v & -2z & & \leq & 6
\end{array}$$

The interval Fourier elimination of x *results in*

$$\begin{array}{cccccc}
\widehat{\mathbf{1/3}}y & & +\widehat{\mathbf{2/3}}z & +u & \leq & \widehat{\mathbf{2}} \\
 & \widehat{-\mathbf{1/3}}v & +\ \widehat{-\mathbf{2/3}}z & +u & \leq & \widehat{\mathbf{3}}
\end{array}$$

The interval Fourier elimination of z result in

$$\widehat{\mathbf{1/3}}y \quad + \widehat{\mathbf{-1/3}}v \quad +\widehat{\mathbf{0}}z \quad +2u \quad \leq \quad \widehat{\mathbf{5.0}}$$

Note that z is not actually eliminated!

We can use quasi-zero elimination to safely eliminate z obtaining: $\widehat{\mathbf{1/3}}y + \widehat{\mathbf{-1/3}}v + 2u \leq \widehat{\mathbf{5.0}} - \widehat{\mathbf{0}} \times [-10, 10]$ *where the right hand side interval has been slightly weakened. This allow us to reduce the domain of u to* $[-10, 5]$.

We can go back to integers by rounding eveything down to arrive at $0y - 1v + 2u \leq 15$, *which only reduces the domain of u to* $[-10, 7]$. *Again we can do better if we first multiply the coefficients by 1000, we obtain* $333y - 334v + 2000u \leq 5010$, *which reduces the domain of u to* $[-10, 5]$.

4.2 A partial Fourier-elimination

Fourier elimination is exponential in the number of generated inequalities in general, in particular intermediate inequalities. In order to avoid this behavior, we perform a Fourier-like elimination on a finite size pool of inequalities. Given a set of inequality constraints $\mathcal{C}$, and a variable x_i, we generate $\mathcal{C}_i$ by applying a normal Fourier elimination step. Then, we keep at most k inequalities from $\mathcal{C} \cup \mathcal{C}_i$ as well as the original ones. We iterate on all variables.

Using a greedy approach to generate linear propagators from a partial Fourier elimination, we need a measure of the propagation effectiveness of the generated inequalities. Recall that the effect of propagator for the inequality $c \equiv \sum a_i x_i \leq a_0$ on x_i is:

$$\underline{d}'_i \leq max(\frac{a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j}{a_i})$$

if $a_i > 0$, and

$$\overline{d}'_i \geq min(\frac{a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j}{a_i})$$

if $a_i < 0$. We can measure the propagation strength of the c for the variable x_i by using the initial domains d^{init} of the variables. Hence the *propagation strength* of c is $max(\frac{a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j^{init}}{a_i})$ if $a_i > 0$ and similarly $min(\frac{a_0 - \sum_{j \neq i} a_j \boldsymbol{d}_j^{init}}{a_i})$ if $a_i < 0$.

Our heuristic for generating strong inequalities through Fourier elimination is as follows. We begin with the initial g inequalities as the pool. We measure the propagation strength og each inequality for each variable x_i and record the inequality and strength of the strongest inequality for each variable in each direction (lower bounding and upper bounding).

We try eliminating each variable x_i in turn by Fourier elimination from the pool. For each generated constraint we measure the propagation strength on each variable, and if better than the current recorded strength, we replace the inequality and strength for this variable and direction. After trying to eliminate each variable from the current pool, the pool becomes the $2n$ inequalities that give the best propagation strength for some variable and direction.

The algorithm repeats until the pool stabilises.

Note that since we use the initial domain $\boldsymbol{d}^{init}$ to determine propagation strength, later in search when the domain has tightened considerably the heuristically chosen propagators may not be that effective. Using the generated constraints to reduce domains as we generate them introduced an overhead while providing little to no pruning given the initial domains and this was removed from the algorithm.

5 Experimental results

The finite domain solver and the interval solver used for the experiments are implemented in Mercury [9]. The interval solver uses the Gaol [6] interval library. Static search strategies with a fixed variable selection strategy and fixed value selection strategy were used for all experiments since the purpose is to compare propagation strength. All variations find the same first solution. The tests were performed on a Pentium 4 1.60GHz running Linux(Sarge).

5.1 Gauss-Jordan elimination

We use variations on well known benchmarks which have dense systems of linear equalities. `eq10` is a well known FD benchmark, involving 7 variables and 10 linear equalities. `magic-square-n` is a scalable arithmetic puzzle involving $2n+2$ linear equalities, n^2 variables and an `alldifferent` constraint. `alpha`, `crypta`, `crypta-magic-sqr`, and `crypta-magic-sqr2` are cryptarithmetic puzzles instances involving an `alldifferent` constraint and respectively 20, 3, 7, and 8 linear equalities and 26, 10, 10, and 11 variables. `overlap-a` is a problem consisting of 3 copies of `alpha` sharing few variables. The problem is handled as a whole system of equations whereas in `partial-overlap-a` each `alpha` copy is pre-processed separately. `alpha-rev` is the same as `alpha` but the labeling order of the variables is the inverse of the original one. `ineq-alpha-rev` is the same problem as `alpha-rev` with 3 redundant linear inequalities involving many of the variables (9,9 and 8) added. In `ineq-alpha-rev` Gauss-Jordan elimination is only performed on equalities whereas it is also performed on inequalities for `ineq-alpha-rev2`.

We compare 5 versions of the problems: `fd` finite domain propagation `fd` on the original constraints; `ic` finite domain propagation with the addition of preconditioned interval constraints; `ic-qz` with the addition of preconditioned interval constraints after quasi-zero elimnination; `ii` finite domain propagaion with the addition of integer inequalities arising from the preconditioned interval constraints; and `ii-qz`) with the addition of integer inequalities arising from the preconditioned interval constraints after quasi-zero elimination. Table 1 and Table 2 shows the comparative times (including all preconditioning and transformation times) and the number of failures for finding the first solution for each of the various methods. We use a fixed labeling order to ensure that the searches are only modified by better pruning. The entry $+\infty$ indicates no solution in 15 minutes.

	Times(ms)				
Problem	fd	ic	ic-qz	ii	ii-qz
eq10	23	10	9	14	11
alpha	277	34	18	15	14
alpha-rev	$+\infty$	79790	28600	5650	5500
ineq-alpha-rev	$+\infty$	79320	29050	6020	5900
ineq-alpha-rev2	$+\infty$	7110	3390	450	430
overlap-a	2470	540	290	270	270
partial-overlap-a	2470	95	55	48	44
crypta	2	6	5	4.4	4
crypta-magic-sqr	78	4	4	2	3
crypta-magic-sqr2	39	4	4	4	4
magic-square-5	408	7250	930	440	450

Table 1: Gauss-Jordan elimination: comparative execution times

	Failures				
Problem	fd	ic	ic-qz	ii	ii-qz
eq10	54	0	0	0	0
alpha	9726	4	4	4	4
alpha-rev	$+\infty$	51639	54098	53709	53709
ineq-alpha-rev	$+\infty$	51214	53914	52380	52380
ineq-alpha-rev2	$+\infty$	3705	6069	3705	3705
overlap-a	77980	4	4	4	4
partial-overlap-a	77981	4	4	4	4
crypta	85	81	81	81	81
crypta-magic-sqr	2946	9	9	9	9
crypta-magic-sqr2	1880	2	2	2	2
magic-square-5	8207	4339	8207	8207	8207

Table 2: Gauss-Jordan elimination: comparative numbers of failures

One can see that the time is substantially reduced as soon as we use add preconditioned interval constraints, and quasi-zero elimination can also give dramatic improvements. The move to integer inequalities is always worthwhile, except for the smallest example `eq10` where the overhead of three propagators for one original constraint is not repaid. Note the mapping to integer inequalities reduces the benefits of quasi-zeros elimination. `ineq-alpha-rev2` shows that applying the approach to inequalities is also beneficial. `partial-overlap-a` illustrates that handling dense subsystems independently can be worthwhile.

Our approach is advantageous when the system after preconditioning has not too many more non-(quasi-)zero coefficients than the original system. Then the better pruning can substantially reduce search. In the case of non-dense systems such as `magic-square-n`, the resulting systems have many more non-zero coefficients, and create weak propagators that gain no benefit and may cost significantly.

5.2 Fourier elimination

We were unable to find standard FD problems with collections of dense linear inequalities with both positive and negatice coefficients (which is required for Fourier elimination). To test partial Fourier elimination we use a set of dense problems with randomly generated coefficients. For all problems, variables take values in the range $[-10, 10]$. s is the seed value used to initialize the random number generator, n is the number of variables, g the number of inequalities, $[l, h]$ the range of the coefficients. $[dl, dh]$ is the range of the distance between an inequality and a randomly generated solution. Thus dl must be positive to guarantee than the generated problem has a solution. For optimization problems, the first sequence of coefficients represent the objective function rather than a regular inequality.

5.2.1 Effect of Fourier elimination

We compare standard finite domain propagation on the original system (`fd`) versus using the same solver on the system resulting by performing a partial interval Fourier elimination to generate interval linear constraints which are then transformed to finite domain linear constraints (`fourier`).

Table 3 compares times (all Fourier elimination preprocessing time is included in the time reported for `fourier`) and failures with and without Fourier preprocessing for instances with at least 1000 failures for the basic finite domain propagator. Note that preprocessing can only reduce the number of failures. Some of the generated problems were trivially solved without preprocessing. For these cases, preprocessing generally does not reduce the number of failures. On the other hand, the overhead of the $2n$ new inequalities slows down the system by a factor of around 2 to 4. When the problems are hard enough, preprocessing the system is nearly always beneficial for this set of problems. The stronger propagation caused by the added inequality propagators reduces the search space enough to compensate for the overhead of running $2n$ linear inequalities propagators in addition to the g original ones.

	Times(ms)		Failures	
Problems	fd	fourier	fd	fourier
s7_n8_g4_l-1_h1_dl10_dh20	40	40	2567	1279
s5_n8_g8_l-1_h1_dl10_dh20	2720	2000	199256	74056
s9_n8_g8_l-1_h1_dl10_dh20	334510	10	24882113	18
s4_n12_g6_l-3_h3_dl20_dh40	2250	150	62266	2044
s6_n12_g6_l-3_h3_dl20_dh40	8440	7850	471987	256601
s7_n12_g6_l-3_h3_dl20_dh40	1950	1580	66457	23524
s8_n12_g6_l-3_h3_dl20_dh40	110	40	6394	1388
s9_n12_g6_l-3_h3_dl20_dh40	530	470	16752	4015
s1_n12_g6_l-1_h1_dl5_dh10	2300	670	168268	17979
s1_n12_g6_l-1_h1_dl5_dh20	80	190	5380	4499
s3_n12_g6_l-1_h1_dl5_dh10	50	20	2813	302
s3_n12_g6_l-1_h1_dl5_dh20	30	20	1099	406

Table 3: Fourier elimination: comparative times and failures

	Failures	
Problems	random	fourier
s7_n8_g4_l-1_h1_dl10_dh20	1279	1279
s5_n8_g8_l-1_h1_dl10_dh20	139574	74056
s9_n8_g8_l-1_h1_dl10_dh20	18	18
s4_n12_g6_l-3_h3_dl20_dh40	53930	2044
s6_n12_g6_l-3_h3_dl20_dh40	238688	256601
s7_n12_g6_l-3_h3_dl20_dh40	54294	23524
s8_n12_g6_l-3_h3_dl20_dh40	6394	1388
s9_n12_g6_l-3_h3_dl20_dh40	15294	4015

Table 4: Failures with our heuristic or a random one

5.2.2 Evaluating the heuristic

At each step of elimination, we keep at most $2n$ inequalities, by using our heuristic for each bound of the n variables. The number of inequalities kept may be less than $2n$ as the same inequality can be the best for two different heuristics. In Table 4 we compare the number of failures between using this heuristic (`fourier`) to select the inequalities versus selecting an inequality randomly for each bound from those that can enforce the bound (`random`). This random inequality is chosen among the inequalities with positive i-th coefficient for the upper bound of the i-th variable and negative coefficient for the lower bound of the i-th variable. The failures alone are given since the times are tightly related to the failures, as the number of constraints in each case is (almost) the same.

Clearly the heuristic achieves much better results in general than a random choice. As it is only a heuristic, there are a few cases where a random choice may give better results.

6 Conclusion

We have shown how we can use Gauss-Jordan elimination and Fourier elimination to improve the propagation of linear constraints. Our approach offers substantial benefits when propagating dense linear systems. Correctness is guaranteed since we only generate safe redundant constraints. Mapping back to integer constraints allows to use our preconditioning techniques with existing optimized finite domain solvers, while the preconditioning only requires a small set of interval arithmetic operations to be available.

There are a number of avenues of further work. Knowing when the elimination approaches will lead to significant gain is an interesting question. For Gauss-Jordan elimination this is sometimes clear by examining density, for Fourier elimination it is less obvious. The example `partial-overlap-a` shows that it is worth finding independent dense subsystems, so detecting these efficiently is clearly of interest. Developing good heuristics for choosing dense subsystems for elimination is clearly an interesting topic for further research.

In this work we have restricted attention to the case when preconditioning is applied once before search begins. Clearly as search progresses the problem can change its neature drastically by fixing variables and reducing domains of variables. It may well be worth preconditioning later in the search, because the remainng constraints may ave become more dense and/or the domains of variable may have shrunk so that the inequalities that propagate most strongly could have changed substantially. This is another avenue that coudl do with further investigation.

Acknowledgments

We would like to thank the referees whose detailed comments have substantially improved this article.

References

[1] C. K. Chiu and J. H. M. Lee. Interval linear constraint solving using the preconditioned interval gauss-seidel method. In *Twelfth International Conference on Logic Programming*, pages 17–31. MIT Press, 1995.

[2] C.W. Choi, W. Harvey, J.H.M. Lee, and P.J. Stuckey. Finite domain bounds consistency revisited. In *Proceedings of the Australian Conference on Artificial Intelligence 2006*, volume 4304 of *LNCS*, pages 49–58. Springer-Verlag, 2006.

[3] ECLiPSe. `http://eclipse.crosscoreop.com/eclipse/`.

[4] T. Feydy and P.J. Stuckey. Propagating dense systems of integer linear equations. In *Proceedings 22nd Annual ACM Symposium on Applied Computing*, pages 306–310. ACM Press, 2007.

[5] J. B. J. Fourier. *Reported in: Analyse de travaux de l'Academie Royale des Sciences, pendant l'annee 1824, Partie Mathematique, Histoire de l'Academie Royale de Sciences de l'Institut de France 7 (1827) xlvii-lv. (Partial English translation in: D.A. Kohler, Translation of a Report by Fourier on his work on Linear Inequalities. Opsearch 10 (1973) 38-42.).* 1824.

[6] F. Goualard. Gaol reference manual. `http://sourceforge.net/projects/gaol/`.

[7] E. Hansen and R. I. Greenberg. An interval Newton method. *Applied Mathematics and Computation*, 12:89–98, 1983.

[8] W. Harvey and P.J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2):173–207, 2003.

[9] F. Henderson et al. The mercury language reference manual. `http://www.mercury.cs.mu.oz.au`.

[10] R. Moore. *Interval Arithmetic*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1966.

[11] A. Neumaier. *Interval Methods for Systems of Equations*, volume 37 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, 1990.

[12] OPL Studio. `www.ilog.com/products/oplstudio/`.

[13] K. Shen and J. Schimpf. Eplex: Harnessing mathematical programming solvers for constraint logic programming. In *Proceedings of Principles and Practice of Constraint Programming*, number 3709 in LNCS, pages 622–636. Springer Verlag, 2005.

[14] C. Solnon. Cooperation of LP solvers for solving MILPs. In *Proceedings International Conference on Tools for Artificial Intelligence*, pages 240–247. IEEE Press, 1997.